Partitioning Sparse Deep Neural Networks for Scalable Training and Inference

Gunduz Vehbi Demirci
gunduz.vehbi.demirci@gmail.com
University of Warwick
United Kingdom

Hakan Ferhatosmanoglu
hakan.f@warwick.ac.uk
University of Warwick
United Kingdom

ABSTRACT

The state-of-the-art deep neural networks (DNNs) have significant computational and data management requirements. The size of both training data and models continue to increase. Sparsification and pruning methods are shown to be effective in removing a large fraction of connections in DNNs. The resulting sparse networks present unique challenges to further improve the computational efficiency of training and inference in deep learning. Both the feedforward (inference) and backpropagation steps in stochastic gradient descent (SGD) algorithm for training sparse DNNs involve consecutive sparse matrix-vector multiplications (SpMVs). We first introduce a distributed-memory parallel SpMV-based solution for the SGD algorithm to improve its scalability. The parallelization approach is based on row-wise partitioning of weight matrices that represent neuron connections between consecutive layers. We then propose a novel hypergraph model for partitioning weight matrices to reduce the total communication volume and ensure computational load-balance among processors. Experiments performed on sparse DNNs demonstrate that the proposed solution is highly efficient and scalable. By utilizing the proposed matrix partitioning scheme, the performance of our solution is further improved significantly.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Artificial intelligence**; **Machine learning**; **Distributed computing methodologies**.

KEYWORDS

Scalable Deep Learning, Sparse Deep Neural Networks, Distributed Stochastic Gradient Descent, Hypergraph Partitioning, Sparse Matrix Vector Multiplication

ACM Reference Format:
Gunduz Vehbi Demirci and Hakan Ferhatosmanoglu. 2021. Partitioning Sparse Deep Neural Networks for Scalable Training and Inference. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460372

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460372

1 INTRODUCTION

Deep neural networks (DNNs) have been extensively utilized in computer vision, speech recognition, and natural language processing [12, 21, 40]. The state-of-the-art DNN architectures demand high storage and computational resources due to the large numbers of parameters (i.e., connection weights) trained over huge datasets. For instance, AlexNet [40], Deepface [56], VGG16 [54] and GPT-3 [5] consist of 60M, 120M, 138M and 175B parameters, respectively. As both the number of parameters and the size of training datasets continue to increase, it is essential to develop scalable training and inference solutions.

Neural network pruning and sparsification methods are successfully applied to address the storage and computational challenges of DNNs [19, 24, 35, 42, 46, 55]. These approaches aim at reducing the amount of memory and computation required to propagate values through the network, typically by removing unimportant connections. They improve DNN's efficiency, scalability, and feasibility in practice, especially for dynamic applications with low latency requirements [64]. Research studies demonstrate that DNNs are tolerant to the sparsification process [28, 47]. For instance, removal of 90% of the connections in ResNet-50 [25] incurs only 3% accuracy loss [18], when trained over ImageNet [17].

Stochastic gradient descent (SGD) is a widely used method for training DNNs. To achieve large-scale learning tasks, parallel SGD algorithms for distributed computing systems (e.g., HPC systems, GPU clusters, TPU pods) are considered in the literature [3, 6, 11, 16, 43, 62, 63]. SGD algorithms that exploit sparsity patterns of networks should be developed to attain efficient training of sparse DNNs and retraining of pruned DNNs. Inference (feedforward) and backpropagation phases of SGD involve consecutive matrix-vector multiplications in such a way that the output vector of one layer is fed as input to the next layer. Matrices in each layer store connection weight parameters between neurons and are updated during the course of training. In the case of sparse DNNs, these matrices become sparse so that computations in each layer heavily depend on sparse matrix-vector multiplications (SpMV).

For large-scale sparse DNNs, we introduce a distributed-memory parallel SGD solution based on efficient parallelization of SpMVs performed in feedforward and backpropagation phases. To perform parallel SpMVs in each layer, matrices and input-output vectors are row-wise partitioned among processors. This partitioning strategy achieves model-wise parallelism. This is in contrast to data-parallel approaches which necessitate the entire model to be stored by processors and face high bandwidth costs and memory bottleneck to perform parameter updates [59]. Our solution reduces memory requirements and performs efficient parameter updates via

model-wise parallelization and utilizes sparse point-to-point communication operations to alleviate bandwidth and latency costs.

We then propose a hypergraph model for partitioning matrices to further scale and improve the efficiency of parallel SpMV computations by reducing the communication costs and achieving computational load-balance among processors. The proposed model utilizes partitioning with fixed vertices to correctly encode the communication requirements of processors and dependencies between successive layers. The partitioning objective of minimizing the cut size in the hypergraph directly encodes the minimization of the total communication volume, and load-balancing constraints enable computational balance among processors.

To evaluate the performance of the proposed training solution with the hypergraph partitioning model, we conduct extensive experiments on several sparse DNN models provided by the Sparse Deep Neural Network Graph Challenge [36] and the MNIST database of handwritten digits [41]. Experimental results show that the parallel SpMV-based sparse DNN training algorithm is highly efficient and scalable, and scales to large processor counts, and the proposed hypergraph partitioning model provides further performance improvements and scalability by significantly reducing both the bandwidth and latency costs of communication.

The contributions of the paper are as follows:

- We introduce a distributed memory-parallel SGD algorithm specifically designed for sparse DNNs to achieve model-wise parallelism.
- To improve parallelization efficiency, we propose a novel hypergraph-based sparse DNN partitioning model which reduces communication costs and achieves a computational balance among processors.
- On a set of sparse DNNs from a benchmark comprising realistic representatives of real-world applications, we performed extensive experiments to analyze the scalability and effectiveness of the proposed algorithm and partitioning model.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents preliminaries. Section 4 describes the proposed distributed-memory parallel SpMV-based SGD solution for sparse DNNs. Section 5 describes our hypergraph model for partitioning sparse DNNs. Section 6 presents experimental results for performance evaluation. Finally, Section 7 concludes the paper.

2 RELATED WORK

Efficient parallel SpMV algorithms for distributed-memory and shared-memory systems are developed in the literature [2, 53, 60]. Several graph/hypergraph partitioning models are proposed to improve the performance of parallel SpMV by reducing the communication costs and achieving the load-balance among processors [7, 26, 34, 38]. Existing approaches, however, are suitable mostly for the cases in which an input matrix is repeatedly multiplied by a vector where the sparsity pattern of the input matrix does not change through the iterations. Hence, these partitioning models and parallel SpMV algorithms are not applicable for sparse DNNs, since each layer is associated with a sparse matrix with different nonzero patterns. Research is needed to design solutions that address the challenges introduced by sparse DNNs and improve their performance.

Motivated by the computational advantages and reduced sizes to handle very large data and models, efficient inference computation on sparse DNNs has attracted significant attention [36]. Parallel algorithms for sparse computations on shared-memory systems are recently proposed (e.g., GPUs [4, 27, 50, 58], multiprocessors [15, 48, 51]). Since these approaches implement only inference computation and are not used for training, each input data vector can be independently processed and distributed parallelism can be achieved by just splitting the input dataset and replicating DNN models among multiple compute nodes. Recently, novel tiling strategies for sparse DNNs are developed to utilize dense matrix kernels for GPUs [23].

Data-parallel methods are widely used to achieve scalability via distributed SGD. In these methods, the dataset is partitioned among multiple compute nodes and local portions of the dataset are processed in terms of batches. Additionally, each compute node stores a local copy of the whole DNN model and depending on the implementation, synchronous or asynchronous updates are performed on the model parameters. Data-parallel SGD algorithms necessitate a large volume of communication between processors since whole model parameters are transferred at each iteration. Therefore, to make data-parallel approaches more feasible, the batch size needs to be increased, but larger batch sizes hurt the training performance of the SGD algorithm. Additionally, to alleviate the high communication cost, gradient compression methods are proposed [1, 45]. More recently, FFT-based gradient sparsification and range-based quantization methods are applied together to reduce the communication volume in data-parallel training algorithms [57].

In synchronous data-parallel methods [3, 20, 29, 29, 44, 62], each node computes local gradients independently and all processors collectively perform an All-reduce communication to receive the average of gradients to update its local parameters. Recently, communication algorithms to improve the efficiency of All-reduce operation on NVLink-enabled dense GPU systems are proposed [10]. To achieve efficiency in synchronous SGD, larger batch sizes should be considered, which may result in lower test accuracy. Methods are proposed to reduce the loss of accuracy due to the use of larger batch sizes [20, 61].

The requirement for processors to synchronize gradient updates after processing each batch causes a significant limitation for the scalability of synchronized SGD. Techniques to overlap communication and computation are proposed to reduce the overheads of synchronization [13, 20]. To achieve further performance improvements, asynchronous methods which differ in communication and update rules are proposed [9, 16, 32, 63]. In asynchronous methods, at each step, a master node (i.e., parameter server) receives local gradients from a worker node, and updates global model parameters then sends the updated model to the same worker where worker nodes are served in arbitrary order. Federated learning algorithms [8, 39] are also considered under this category where a subset of clients download the most recent model from a central server and computes updates to the model. Then the clients send their model updates to the central server which aggregates these model updates typically by averaging to improve the global model.

Alternative to data-parallel methods, training approaches that aim at model-wise parallelism are also considered [30, 31]. For example, FlexFlow [31] searches different parallelization strategies

by performing simulations before training. However, this tool is mainly designed for GPU clusters and does not provide a partitioning on sparse DNNs. The proposed model-wise parallelism in our SGD algorithm offers inherent scalability, whereas in data-parallel approaches, each processor holds the whole set of parameters and broadcasts gradients for these parameters to all processors. Therefore, in data-parallel approaches, the total communication volume significantly increases with increasing number of processors, and the local memory size of processors limits the size of neural networks. As validated in the experiments, the proposed SGD algorithm reduces the total communication volume, since each processor only keeps a small set of parameters and broadcasts their gradients to a small subset of processors.

3 PRELIMINARIES

3.1 Hypergraph Partitioning

Let $H = (V, N)$ denote a hypergraph where V and N are vertex and net sets, respectively. Each net $n_j \in N$ may connect multiple vertices and the set of vertices that connected by n_j is represented by pins(n_j). Each vertex $v_i \in V$ is associated with weight $w(v_i)$ and each net $n_j \in N$ is associated with cost(n_j). A P-way partition of H is defined as $\Pi = \{V_1, V_2 \cdots V_P\}$ consisting of mutually disjoint and exhaustive subsets of vertices $V_m \subset V$ where $V_m \cap V_n = \emptyset$ if $m \neq n$ and $V_m \neq \emptyset$ for all $V_m \in \Pi$ such that $\bigcup V_m = V$.

Under a partition Π, a net n_j connects to a part V_m if pins(n_j)$\cap$ $V_m \neq \emptyset$. The set of parts that are connected by net n_j is defined as the connectivity set $\Lambda(n_j)$ and the number of parts that are connected by net n_j is defined as connectivity $\lambda(n_j) = |\Lambda(n_j)|$. A net n_j is said to be cut if it connects to multiple parts (i.e., $\lambda(n_j) > 1$) and uncut otherwise. The connectivity cut size under Π is defined as

$$\chi(\Pi) = \sum_{n_j \in N} \text{cost}(n_j) \times (\lambda(n_j) - 1) \tag{1}$$

The weight of a part $V_m \in \Pi$ is defined as $W(V_m) = \sum_{v_i \in V_m} w(v_i)$. The partition Π is balanced if it satisfies

$$W(V_m) \leq W_{avg}(1 + \epsilon), \quad \forall\, V_m \in \Pi \tag{2}$$

where $W_{avg} = \sum_{v_i \in V} w(v_i)/P$ is the average part weight and ϵ is the maximum allowed imbalance ratio.

The hypergraph partitioning problem for finding a P-way partition with the objective of minimizing the cut size given in (1) and satisfying balancing constraints in (2) is NP-Hard. There exist tools that produce quality results for the hypergraph partitioning problem [7, 33]. These tools also support partitioning hypergraphs with fixed vertices where some vertices can be assigned to parts prior to partitioning.

3.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an optimization technique which is commonly used for training neural networks to iteratively minimize a loss function over an input dataset. SGD is usually implemented in two main phases which heavily depend on SpMVs: (1) Feedforward (inference) phase, (2) Backpropagation phase.

Given a DNN composed of L layers where connection weights in each layer $k = 1, 2, \ldots, L$ are represented by a matrix $\mathbf{W}^k$ such that the connection weight from the ith neuron in layer k to the jth neuron in layer $k+1$ is denoted by nonzero entry $\mathbf{W}^k(j, i)$. In the inference phase, an input vector $\mathbf{x}^0$ is sent through the network layers to compute an output vector $\mathbf{x}^L$. Formally, the inference step can be given as

$$\mathbf{x}^k = f(\mathbf{W}^k \mathbf{x}^{k-1} + \mathbf{b}^k) \tag{3}$$

where $\mathbf{b}^k$ denotes the bias vector and $f(\cdot)$ is a nonlinear activation function applied to each element of a vector. The bias vector $\mathbf{b}^k$ can be embedded in matrix $\mathbf{W}^k$ as the first column and the first entries of vectors $\mathbf{x}^k$ can be set to one (i.e., the number of dimension of $\mathbf{x}^k$ increases by one). In a simpler form, the feedforward computation in each layer k becomes $\mathbf{x}^k = f(\mathbf{W}^k \mathbf{x}^{k-1})$.

In the backpropagation phase, output vector $\mathbf{x}^L$ of the inference step is used for computing gradient vector $\boldsymbol{\delta}^L$ which is backpropagated to compute gradients $\boldsymbol{\delta}^k$ in preceding layers $k = 1, 2, \ldots, L$–1. The ith component of vector $\boldsymbol{\delta}^k(i)$ denotes the partial derivative of a loss function $\mathbf{J}(\mathbf{x}^L, \mathbf{y})$ with respect to the total input activation of the ith neuron in layer k. Vector $\mathbf{y}$ is the true label for input vector $\mathbf{x}^0$ where the loss function depends on both of the vectors. Each gradient $\boldsymbol{\delta}^k$ is used to update weight matrix $\mathbf{W}^k$ by the following gradient update rule

$$\frac{\partial \mathbf{J}}{\partial \mathbf{W}^k(j, i)} = \boldsymbol{\delta}^k(j)\, \mathbf{x}^{k-1}(i) \tag{4}$$

$$\mathbf{W}^k(j, i) \leftarrow \mathbf{W}^k(j, i) - \eta \frac{\partial \mathbf{J}}{\partial \mathbf{W}^k(j, i)} \tag{5}$$

where η denotes the learning rate. The gradient vector $\boldsymbol{\delta}^L$ in the final layer L is computed as

$$\boldsymbol{\delta}^L = \nabla_{\mathbf{x}^L} \mathbf{J} \odot f'(\mathbf{z}^L) \tag{6}$$

where $\nabla_{\mathbf{x}^L} \mathbf{J}$ is a vector of derivatives of the loss function $\mathbf{J}$ with respect to the outputs of the activation functions in the final layer (i.e., $\mathbf{x}^L$) and $f'(\mathbf{z}^L)$ is the vector of derivatives of the outputs with respect to the input activation $\mathbf{z}^L = \mathbf{W}^L \mathbf{x}^L$ (i.e., local gradients) in layer L and symbol "$\odot$" denotes element-wise multiplication. Gradients for layers $k = 1, 2, \ldots, L-1$ are computed by a recursive formula

$$\boldsymbol{\delta}^{k-1} = (\mathbf{W}^k)^T \boldsymbol{\delta}^k \odot f'(\mathbf{z}^{k-1}). \tag{7}$$

Algorithm 1 SGD

Require: T, $\{\mathbf{W}^k\}$
1: **for each** $\mathbf{x}^0 \in T$ **do**
2: **for** $k = 1, 2, \ldots, L$ **do**
3: $\mathbf{z}^k = \mathbf{W}^k \mathbf{x}^{k-1}$
4: $\mathbf{x}^k = f(\mathbf{z}^k)$
5: $\boldsymbol{\delta}^L = \nabla_{\mathbf{x}^L} J \odot f'(\mathbf{z}^L)$
6: **for** $k = L, L-1, \ldots, 1$ **do**
7: $\boldsymbol{\delta}^{k-1} = (\mathbf{W}^k)^T \boldsymbol{\delta}^k \odot f'(\mathbf{z}^{k-1})$
8: $\nabla \mathbf{W}^k = \boldsymbol{\delta}^k \otimes \mathbf{x}^{k-1}$
9: $\mathbf{W}^k \leftarrow \mathbf{W}^k - \eta \nabla \mathbf{W}^k$

Algorithm 1 displays the overall execution of SGD. The for loop in lines 1–9 is executed overall input vectors in training dataset T in such a way that for each input vector $\mathbf{x}^0 \in T$, feedforward and backpropagation steps are executed. Lines 2–4 correspond to

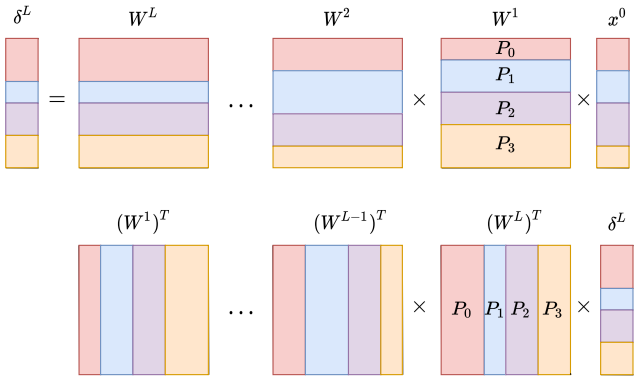

Figure 1: Illustration of sequences of SpMVs performed in SpFF (Top) and SpBP (Bottom).

inference (feedforward) step where repeated SpMVs of the form $\mathbf{z}^k = \mathbf{W}^k\mathbf{x}^{k-1}$ are performed. Between two consecutive layers, nonlinear activation function $f(\cdot)$ is applied to each component of vector $\mathbf{z}^k$ and the output vector $f(\mathbf{z}^k)$ of layer k is fed as input to the next layer $k+1$. In line 5, the gradient vector $\boldsymbol{\delta}^L$ is computed using the output vector $\mathbf{x}^L$ and the input activations $\mathbf{z}^L$ of the final layer L. Lines 6–9 correspond to backpropagation step where repeated SpMVs of the form $\boldsymbol{\delta}^{k-1} = (\mathbf{W}^k)^T\boldsymbol{\delta}^k$ are performed to backpropagate gradient vectors. In line 8, outer product of gradient vector $\boldsymbol{\delta}^k$ with vector $\mathbf{x}^{k-1}$ is performed to produce matrix $\nabla\mathbf{W}^k$ which is used to update weight matrix $\mathbf{W}^k$ by the gradient update rule.

4 DISTRIBUTED SGD ALGORITHM FOR SPARSE DNNS

In order to achieve a parallel training algorithm (i.e., parallel SGD) for sparse DNNs, we develop parallel SpMV-based feedforward and backpropagation algorithms in Sections 4.1 and 4.2, respectively. That is, parallel sparse feedforward (SpFF) algorithm achieves parallelization of lines 2–4, whereas parallel sparse backpropagation (SpBP) algorithm achieves parallelization of lines 5–9 in Algorithm 1.

Figure 1 displays the general execution of the parallel SGD algorithm together with its weight matrix partitioning scheme. In the figure, only sequences of SpMV operations are displayed whereas the remaining computations are omitted for ease of exposition. In the inference phase, input vector $\mathbf{x}^0$ and weight matrices $\mathbf{W}^1, \mathbf{W}^2 \dots \mathbf{W}^L$ are row-wise partitioned among four processors. For each layer $k = 1, 2, \dots L$, processors perform communication to receive nonlocal entries of vector $\mathbf{x}^{k-1}$ and collectively perform SpMV $\mathbf{W}^k\mathbf{x}^{k-1}$ to compute vector $\mathbf{x}^k$. The output vector $\mathbf{x}^k$ computed in layer k is used as input in the next layer. This process is repeatedly performed until the final layer L where the gradient vector $\boldsymbol{\delta}^L$ is computed. In the backpropagation phase, gradient vector $\boldsymbol{\delta}^L$ is row-wise partitioned among processor whereas transposes of weight matrices $(\mathbf{W}^1)^T, (\mathbf{W}^2)^T \dots (\mathbf{W}^L)^T$ are column-wise partitioned. The row-wise partitioning of weight matrices induces column-wise partitioning on their transposes. For each layer $k = L, L-1, \dots 1$, processors collectively perform SpMV $(\mathbf{W}^k)^T\boldsymbol{\delta}^k$. Here, since matrices are column-wise partitioned, processors communicate partial products contributing to the same nonzero entries of output vector $\boldsymbol{\delta}^{k-1}$ instead of communicating entries of input vector $\boldsymbol{\delta}^k$. These partial products are summed by processors to get the final values of entries in gradient vector $\boldsymbol{\delta}^{k-1}$ which is used as input in the next layer.

Algorithm 2 SpFF

Require: $\mathbf{x}^0_m$, $\{\mathbf{W}^k_m\}$, $\{\text{Xsend}^k_m\}$, $\{\text{Xrecv}^k_m\}$
1: **for all** *processors* P_m **in parallel do**
2: **for** $k = 1, 2, \dots, L$ **do**
3: **for each** $(P_n, \bar{\mathbf{x}}^{k-1}_{mn}) \in \text{Xsend}^k_m$ **do**
4: Update $\bar{\mathbf{x}}^{k-1}_{mn}$ with entries in $\mathbf{x}^{k-1}_m$
5: Non-blocking send $\bar{\mathbf{x}}^{k-1}_{mn}$ to P_n
6: $\mathbf{z}^k_m = \mathbf{W}^k_m\mathbf{x}^{k-1}_m$
7: **for each** $(P_n, \hat{\mathbf{x}}^{k-1}_{nm}) \in \text{Xrecv}^k_m$ **do**
8: Receive nonzero entries $\hat{\mathbf{x}}^{k-1}_{nm}$ from process P_n
9: $\mathbf{z}^k_m \leftarrow \mathbf{z}^k_m + \mathbf{W}^k_m\hat{\mathbf{x}}^{k-1}_{nm}$
10: $\mathbf{x}^k_m = f(\mathbf{z}^k_m)$

4.1 Parallel Sparse Feedforward

The parallel sparse feedforward (SpFF) performs repeated parallel SpMV in the form of $\mathbf{W}^k\mathbf{x}^{k-1}$ for each layer $k = 1, 2, \dots, L$. Parallelism is achieved through row-wise partitioning of weight matrices $\mathbf{W}^k$ and input/output vectors $\mathbf{x}^{k-1}$ among processors.

Algorithm 2 displays the overall execution of the proposed SpFF algorithm. In the algorithm, each processor P_m for $m = 1, 2, \dots, P$ stores row-blocks $\mathbf{W}^k_m$ and $\mathbf{x}^{k-1}_m$ of matrix $\mathbf{W}^k$ and vector $\mathbf{x}^{k-1}$, respectively. Additionally, each processor P_m is provided with maps Xsend^k_m and Xrecv^k_m that map row indices of vector $\mathbf{x}^{k-1}_m$ to processor ids. In this way, each processor knows which $\mathbf{x}^{k-1}$-vector entries to be communicated with which processor. Formally, these sets are defined as

$$\text{Xsend}^k_m = \left\{(P_n, \bar{\mathbf{x}}^{k-1}_{mn}) \mid \bar{\mathbf{x}}^{k-1}_{mn} = \mathbf{x}^{k-1}_m[cols(\mathbf{W}^k_n) \cap rows(\mathbf{W}^{k-1}_m)]\right\} \tag{8}$$

$$\text{Xrecv}^k_m = \left\{(P_n, \hat{\mathbf{x}}^{k-1}_{nm}) \mid \hat{\mathbf{x}}^{k-1}_{nm} = \mathbf{x}^{k-1}_n[cols(\mathbf{W}^k_m) \cap rows(\mathbf{W}^{k-1}_n)]\right\} \tag{9}$$

where $cols(\cdot)$ and $rows(\cdot)$ respectively denote the indices of columns and rows that contain at least one nonzero entry in a given matrix/vector. $\mathbf{x}^{k-1}_m[\cdot]$ and $\mathbf{x}^{k-1}_n[\cdot]$ denote subvectors that are composed of given row indices of vectors $\mathbf{x}^{k-1}_m$ and $\mathbf{x}^{k-1}_n$, respectively. Hence, for each $(P_n, \bar{\mathbf{x}}^{k-1}_{mn}) \in \text{Xsend}^k_m$, processor P_m sends subvector $\bar{\mathbf{x}}^{k-1}_{mn}$ to processor P_n whereas for each $(P_n, \hat{\mathbf{x}}^{k-1}_{nm}) \in \text{Xrecv}^k_m$, processor P_m receives subvector $\hat{\mathbf{x}}^{k-1}_{nm}$ from processor P_n.

Sets Xsend^k and Xrecv^k are precomputed by using the sparsity patterns of weight matrices (i.e., neuron connections) and the row partitioning of weight matrices among processors. The row-wise

partitioning of weight matrices induces neuron partitioning in each layer so that all computations related to a neuron are performed by a single processor. As shown in (8) and (9), to perform $\mathbf{W}_m^k \mathbf{x}^{k-1}$, processor P_m needs to receive all $\mathbf{x}^{k-1}$-vector rows corresponding to column indices in $cols(\mathbf{W}_m^k)$. It is important to note that vectors $\bar{\mathbf{x}}_{mn}^{k-1}$ and $\hat{\mathbf{x}}_{nm}^{k-1}$ are placeholders that keep coordinates of nonzero entries. Hence, nonzero entries of these vectors are updated before used in any operation. For instance, before sending to processor P_n, nonzero entries of vector $\bar{\mathbf{x}}_{mn}^{k-1}$ are updated (i.e., line 4) with the corresponding entries in $\mathbf{x}_m^{k-1}$ locally computed in the preceding layer. Similarly, nonzero entries of vector $\hat{\mathbf{x}}_{nm}^{k-1}$ must be received from processor P_n before it is multiplied by weight matrix $\mathbf{W}_m^k$ (i.e., lines 8–9).

In the algorithm, for each layer $k = 1, 2, \ldots, L$, the for loop in line 2 is executed in parallel by all processors: In lines 3–5, each processor P_m performs a non-blocking communication for each tuple $(P_n, \bar{\mathbf{x}}_{mn}^{k-1}) \in \text{Xsend}_m^k$ to send its local nonzero entries in $\bar{\mathbf{x}}_{mn}^{k-1}$ to processor P_n. To overlap communication by computation, each processor performs local SpMV computation $\mathbf{z}_m^k = \mathbf{W}_m^k \mathbf{x}_m^{k-1}$ without waiting for the messages to be received by recipient processors. Entries of $\mathbf{z}^k$ store the total activation values incoming to neurons. For instance, nonzero entry $\mathbf{z}^k(i)$ stores the total activation of the ith neuron in layer k. After local SpMV computations are performed, for each tuple $(P_n, \hat{\mathbf{x}}_{nm}^{k-1}) \in \text{Xrecv}_m^k$, processor P_m receives vector $\hat{\mathbf{x}}_{nm}^{k-1}$ from processor P_n and multiplies by $\mathbf{W}_m^k$ to update the corresponding entries in vector $\mathbf{z}_m^k$ (i.e., lines 7–9). Finally, a nonlinear activation function (i.e., ReLu, sigmoid etc.) is applied to $\mathbf{z}_m^k$ and the respective output elements in $\mathbf{x}_m^k$ are computed.

4.2 Parallel Sparse Backpropagation

The parallel sparse backpropagation (SpBP) works similarly to SpFF algorithm where SpBP performs repeated SpMVs in the form of $\boldsymbol{\delta}^{k-1} = (\mathbf{W}^k)^T \boldsymbol{\delta}^k$ in the reverse order that of performed by SpFF. Since the weight matrices are row-wise partitioned among processors, each processor P_m for $m = 1, 2, \ldots, P$ stores column-block $(\mathbf{W}_m^k)^T$ of matrix $(\mathbf{W}^k)^T$ and row-block $\boldsymbol{\delta}_m^k$ of gradient vector $\boldsymbol{\delta}^k$, respectively. Therefore, each processor P_m multiplies its local gradient vector $\boldsymbol{\delta}_m^k$ by transpose $(\mathbf{W}_m^k)^T$ of its local weight matrix $\mathbf{W}_m^k$ in each layer k.

Algorithm 3 displays the overall execution of SpBP. As a first step, each processor P_m locally computes gradient vector $\boldsymbol{\delta}_m^L$ according to Eq. (6) by using the output vector $\mathbf{x}_m^L$ computed in the inference phase. By executing the for loop in lines 3–13 in parallel, vector $\boldsymbol{\delta}^L$ is backpropagated through the layers $L, L-1, \ldots, 1$. To backpropagate vector $\boldsymbol{\delta}^k$ to the preceding layer k–1, an SpMV of the form $\mathbf{s}_m^k = (\mathbf{W}_m^k)^T \boldsymbol{\delta}_m^k$ is performed in line 4. Vector $\mathbf{s}_m^k$ may contain partial derivatives contributing to neuron outputs computed on different processors as well as to local neuron outputs. Nonzeros of vector $\mathbf{s}_m^k$ that are contributing to neurons located on different processors are sent to the corresponding processors. Nonzeros that are contributing to the local neuron outputs are summed with the partial derivatives received from other processors, before multiplying with local gradients $f'(\mathbf{z}^{k-1})$. That is, communication operations are performed on nonzero entries of $\mathbf{s}_m^k$.

Algorithm 3 SpBP

Require: $\mathbf{x}_m^L$, $\{(\mathbf{W}_m^k)^T\}$, $\{\text{Ssend}_m^k\}$, $\{\text{Srecv}_m^k\}$
1: **for all** *processors* P_m **in parallel do**
2: $\quad \boldsymbol{\delta}_m^L = \nabla_{\mathbf{x}_m^L} \mathbf{J} \odot f'(\mathbf{z}_m^L)$
3: $\quad$ **for** $k = L, L-1, \ldots, 1$ **do**
4: $\quad\quad \mathbf{s}_m^k = (\mathbf{W}_m^k)^T \boldsymbol{\delta}_m^k$
5: $\quad\quad$ **for each** $(P_n, \bar{\mathbf{s}}_{mn}^k) \in \text{Ssend}_m^k$ **do**
6: $\quad\quad\quad$ Update $\bar{\mathbf{s}}_{mn}^k$ with corresponding entries in $\mathbf{s}_m^k$
7: $\quad\quad\quad$ Non-blocking send $\bar{\mathbf{s}}_{mn}^k$ to P_n
8: $\quad\quad \nabla \mathbf{W}_m^k = \boldsymbol{\delta}_m^k \otimes \mathbf{x}_m^{k-1}$
9: $\quad\quad \mathbf{W}_m^k \leftarrow \mathbf{W}_m^k - \eta \nabla \mathbf{W}_m^k$
10: $\quad\quad$ **for each** $(P_n, \hat{\mathbf{s}}_{nm}^k) \in \text{Srecv}_m^k$ **do**
11: $\quad\quad\quad$ Receive nonzero entries $\hat{\mathbf{s}}_{nm}^k$ from process P_n
12: $\quad\quad\quad \mathbf{s}_m^k \leftarrow \mathbf{s}_m^k + \hat{\mathbf{s}}_{nm}^k$
13: $\quad\quad \boldsymbol{\delta}_m^{k-1} = \mathbf{s}_m^k \odot f'(\mathbf{z}_m^{k-1})$

As in SpFF algorithm, each processor is provided with maps Ssend^k and Srecv^k, where for each tuple $(P_n, \bar{\mathbf{s}}_{mn}^k) \in \text{Ssend}_m^k$ there exists $(P_n, \hat{\mathbf{x}}_{nm}^{k-1}) \in \text{Xrecv}_m^k$ and $rows(\bar{\mathbf{s}}_{mn}^k) = rows(\hat{\mathbf{x}}_{nm}^{k-1})$. Similarly, for each tuple $(P_n, \hat{\mathbf{s}}_{nm}^k) \in \text{Srecv}_m^k$ there exists $(P_n, \bar{\mathbf{x}}_{mn}^{k-1}) \in \text{Xsend}_m^k$ and $rows(\hat{\mathbf{s}}_{nm}^k) = rows(\bar{\mathbf{x}}_{mn}^{k-1})$. That is, if processor P_m receives a nonzero $\mathbf{x}^{k-1}(i)$ from processor P_n, then P_m sends the corresponding gradient contribution $\mathbf{s}^k(i)$ to P_n. Similarly, if processor P_m sends a nonzero $\mathbf{x}^{k-1}(j)$ to processor P_n, then P_m receives the corresponding gradient contribution $\mathbf{s}^k(j)$ from P_n.

In lines 5–7, each processor P_m performs a non-blocking communication for each tuple $(P_n, \bar{\mathbf{s}}_{mn}^k) \in \text{Ssend}_m^k$ to send nonzero entries in $\bar{\mathbf{s}}_{mn}^k$ to processor P_n. To overlap communication by computation, each processor locally performs outer product $\boldsymbol{\delta}_m^k \otimes \mathbf{x}_m^{k-1}$ without waiting for the messages to be received by recipient processors. It is important to note that $\mathbf{x}_m^{k-1}$ contains nonzero entries received from other processors in the inference phase. The outer product produces matrix $\nabla \mathbf{W}_m^k$ which is used to update weight matrix $\mathbf{W}_m^k$ in lines 8–9. After updating weight matrices, for each tuple $(P_n, \hat{\mathbf{s}}_{nm}^k) \in \text{Srecv}_m^k$, processor P_m receives nonzero entries in $\hat{\mathbf{s}}_{nm}^k$ from processor P_n and sums the received nonzero entries with the corresponding entries in $\mathbf{s}_m^k$ to compute the final partial derivatives for the local neuron outputs (i.e., lines 10–12). Finally, nonzero entries of $\mathbf{s}_m^k$ are multiplied with local gradients in line 13 and gradient vector $\boldsymbol{\delta}_m^{k-1}$ for the preceding layer $k-1$ is computed. It is important to highlight that only the nonzero entries of $\mathbf{s}_m^k$, which correspond to $rows(\mathbf{x}_m^{k-1})$, are multiplied by local gradients $f(\mathbf{z}_m^{k-1})$ and carried into vector $\boldsymbol{\delta}_m^{k-1}$.

5 HYPERGRAPH PARTITIONING MODEL FOR SPARSE DNNS

We propose a hypergraph model for partitioning rows of weight matrices (i.e., neural network) among processors to optimize communication costs of parallel SpMV operations performed by SpFF and SpBP algorithms. The proposed model adopts a multi-phase and fixed vertex partitioning approach to correctly encode communication patterns of processors between consecutive layers.

Our partitioning model consists of L phases ϕ^k for $k = 1, 2, \ldots, L$. In each phase ϕ^k, rows of matrix $\mathbf{W}^k$ are partitioned into P parts. Note that the row-wise partitioning of weight matrix $\mathbf{W}^k$ induces column-wise partitioning of $(\mathbf{W}^k)^T$ in backpropagation phase. For each phase ϕ^k, we define a hypergraph $H(\phi^k) = (V^k \cup F^k, N^k)$, where for each matrix row $\mathbf{W}^k(i,:)$, there exists one vertex $v_i \in V^k$, for each column $\mathbf{W}^k(:,j)$, there exists one fixed vertex $v_j^f \in F^k$ and one net $n_j \in N^k$.

Each vertex $v_i \in V^k$ represents row $\mathbf{W}^k(i,:)$ (i.e., the ith neuron) and all computations associated with that row. In the inference phase, vertex v_i represents the task of computing the inner product

$$\mathbf{z}^k(i) = \mathbf{W}^k(i,:)\mathbf{x}^{k-1} = \sum_{\mathbf{W}^k(i,j) \in \mathbf{W}^k(i,:)} \mathbf{W}^k(i,j)\mathbf{x}^{k-1}(j) \tag{10}$$

which corresponds to the computation of the ith neuron's total input activation. In the backpropagation phase, vertex v_i represents column $(\mathbf{W}^k)^T(:,i)$ and the task of computing multiplications in sparse SAXPY/DAXPY operations $\mathbf{s}^k(j) = \mathbf{s}^k(j) + (\mathbf{W}^k)^T(j,i)\boldsymbol{\delta}(i)^k$ for each nonzero row index j in column $(\mathbf{W}^k)^T(:,i)$. Additionally, vertex v_i also represents gradient update operations

$$\mathbf{W}^k(i,:) \leftarrow \mathbf{W}^k(i,:) - \eta \nabla \mathbf{W}^k(i,:) \tag{11}$$

associated with the links connected to the ith neuron in layer k. Therefore, each vertex is associated with a computational weight equal to the number of nonzeros in row $\mathbf{W}^k(i,:)$ (i.e., number of links connecting to the ith neuron). Fixed vertices in set F^k do not represent any computation and are introduced to connect nets to prespecified parts for correctly encoding input-output dependencies between consecutive layers in multi-phase partitioning framework.

A P-way partitioning $\Pi_P(\phi^k) = \{V_1^k, V_2^k, \ldots, V_P^k\}$ on hypergraph $H(\phi^k)$ denotes that all tasks corresponding to vertices in part $V_m^k \in \Pi_P(\phi^k)$ are assigned to processor P_m. For instance, if a vertex v_i is assigned to part V_m^k, then processor P_m stores row $\mathbf{W}^k(i,:)$ and performs all computation associated with this row. Partitioning Π_P induces a partial reordering so that the matrix rows belonging to the same part can be reordered consecutively (in any order) to form a row block $\mathbf{W}_m^k$ which is assigned to processor P_m.

Net set N^k simultaneously encodes the total communication volume of processors during inference and backpropagation phases. In the inference phase, each net $n_j \in N^k$ represents the set of tasks (vertices) that need nonzero entry $\mathbf{x}^{k-1}(j)$, whereas in the backpropagation phase, each net n_j represents the set of tasks that contribute to the computation of nonzero entry $\mathbf{s}^k(j)$. Hence, net n_j connects each vertex $v_i \in V^k$ for which the corresponding row $\mathbf{W}^k(i,:)$ has a nonzero entry in the jth column.

In order to satisfy input-output dependencies between successive layers, each net $n_j \in N^k$ connects only one fixed vertex $v_j^f \in F^k$ and fixed vertex v_j^f only connects n_j. Fixed vertex v_j^f represents nonzero $\mathbf{x}^{k-1}(j)$ and it is fixed to the same part/processor to which row $\mathbf{W}^{k-1}(j,:)$ is assigned in the preceding phase ϕ^{k-1}, since $\mathbf{x}^{k-1}(j)$ is locally computed by that processor in layer $k-1$ (i.e., $v_j \in V_m^{k-1} \Rightarrow v_j^f \in V_m^k$). In other words, fixed vertex $v_j^f \in F^k$ ensures that after partitioning in phase ϕ^k, net $n_j \in N^k$ connects the part/processor which is given the responsibility of computing nonzero $\mathbf{x}^{k-1}(j)$. Formally, pins of net n_j is defined as

$$\text{pins}(n_j) = \{v_i \in V^k \mid \exists j \in \text{rows}(\mathbf{W}^k(i,:))\} \cup \{v_j^f\}. \tag{12}$$

In the inference phase, a cut net $n_j \in N^k$ whose fixed vertex v_j^f is assigned to a part $V_m^k \in \Lambda(n_j)$ implies that nonzero $\mathbf{x}^{k-1}(j)$ is computed by processor P_m and will be sent from P_m to all processors in $\Lambda(n_j) \setminus V_m^k$. Therefore, cut net n_j incurs the communication volume of $\left(|\Lambda(n_j)| - 1\right)$ words in the kth layer of SpFF. In the backpropagation phase, each processor $P_n \in \Lambda(n_j) \setminus V_m^k$, computes its contribution to nonzero $\mathbf{s}^k(j)$ and sends to processor P_m. Therefore, cut net n_j incurs the communication volume of $\left(|\Lambda(n_j)| - 1\right)$ words in the backpropagation phase as well. As seen here, if a processor P_m sends a nonzero $\mathbf{x}^{k-1}(j)$ to a processor P_n in the inference phase, processor P_m receives the corresponding gradient contribution $\mathbf{s}^k(j)$ from processor P_n. Therefore, the total communication volume between processors during SpFF and SpBP in layer k can be given as

$$\text{Vol}(k) = \sum_{n_j \in N^k} 2 \times \left(|\Lambda(n_j)| - 1\right).$$

Therefore, if each net $n_j \in N^k$ is associated with $cost(n_j) = 2$, the partitioning objective of minimizing the cutsize in phase ϕ^k encodes the minimization of the total communication volume during performance of SpFF and SpBP in layer k. Note that each net is associated with equal $cost(n_j) = 2$ which encodes the number of nonzeros transferred during inference and backpropagation phases. Any uniform cost association is valid for partitioning.

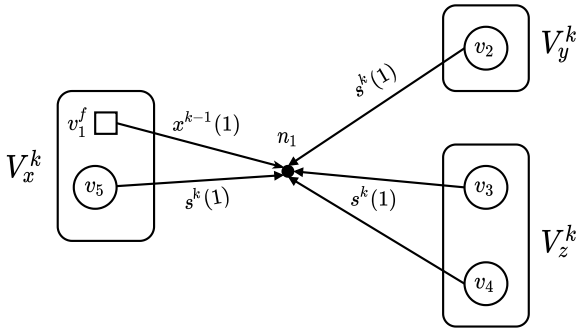

Figure 2: Cut net n_1 encoding communication operations between neurons represented by v_1^f in layer $k-1$ and neurons v_2, v_3, v_4, v_5 in layer k.

Figure 2 displays an illustrative example where cut net n_1 with $\Lambda(n_1) = \{V_x^k, V_y^k, V_z^k\}$ is given. In the figure, fixed vertex v_1^f is preassigned to part V_x^k by partitioning $\Pi(\phi^{k-1})$ so that the task of computing $\mathbf{x}^{k-1}(1)$ is given to processor P_x (i.e., $v_1 \in V_x^{k-1}$). Hence, in the kth step of inference phase, processor P_x sends $\mathbf{x}^{k-1}(1)$ to processors P_y and P_z, since output of the neuron v_1^f is connected to neurons v_2, v_3 and v_4. Here neuron v_5 does not incur communication since it is assigned to the same processor P_x by partitioning $\Pi(\phi^k)$. Even though the output of neuron v_1^f (i.e., neuron v_1 in layer $k-1$) is connected to two neurons v_3 and v_4 in processor

P_y, nonzero $\mathbf{x}^{k-1}(1)$ is sent only once to this processor. So net n_1 encodes a communication volume of $|\Lambda(n_j)| - 1 = 2$ words during SpFF in layer k. Similarly, in the backpropagation phase, processors P_y and P_z send partial gradient contributions for $\mathbf{s}^k(1)$ to processor P_x. Partial gradient contribution of vertex v_5 is locally summed by P_x and does not contribute to the total communication volume. Note that P_z sums partial gradients contributions for each of its vertices v_3 and v_4 before sending a single value to P_x. Hence, as in the inference phase, net n_1 encodes the same communication volume of $|\Lambda(n_j)| - 1 = 2$ words during SpBP in layer k.

Figure 3 displays an illustrative example of the proposed hypergraph partitioning model. The sparse DNN in the top left in the figure consists of three layers each of which contains four neurons (i.e., $\mathbf{x}^0$ corresponds to the input layer). Weight matrices $\mathbf{W}^1$ and $\mathbf{W}^2$ are displayed in the top right of the figure where connections between neurons are denoted by nonzero entries. For instance, neuron 2 in the first layer is represented by row $\mathbf{W}^1(2,:)$ where the columns 1, 2 and 3 have nonzero entries, since neuron 2 connects neurons 1, 2 and 3 in the input layer. The two subfigures of the lower part display hypergraphs $H(\phi^1)$ and $H(\phi^2)$ which contain four vertices and four nets corresponding to rows and columns of matrices $\mathbf{W}^1$ and $\mathbf{W}^2$, respectively. Additionally, $H(\phi^2)$ contains four fixed vertices which correspond to rows of $\mathbf{x}^1$. Fixed vertices v_3^f and v_4^f are preassigned to part V_1^2 whereas v_1^f and v_2^f are preassigned to part V_2^2, since v_3 and v_4 are assigned to V_1^1 whereas v_1 and v_2 are assigned to V_1^1 by $\Pi(\phi^1)$ in the previous layer. That is, nonzeros $\mathbf{x}^1(3)$ and $\mathbf{x}^1(4)$ are computed by the processor P_1 whereas the rows $\mathbf{x}^1(1)$ and $\mathbf{x}^1(2)$ are computed by the processor P_2. Therefore, in layer 2, nonzeros $\mathbf{x}^1(3)$ and $\mathbf{x}^1(4)$ will be sent from P_1 to P_2, and the rows $\mathbf{x}^1(1)$ and $\mathbf{x}^1(2)$ will be sent from P_2 to P_1. In the figure, rows of the input vector $\mathbf{x}^0$ can be assigned to processors with respect to net connectivities. For instance, row $\mathbf{x}^0(1)$ can be stored by one of the processors P_1 and P_2, since net n_1 connects both parts V_1^1 and V_2^1. On the other hand, net n_2 only connects V_2^1 and hence, $\mathbf{x}^0(2)$ is stored locally by P_2 and it is not communicated.

The running time complexity of the partitioning phase depends on the sizes of hypergraphs built in each phase and the partitioning algorithm/tool used. The sizes of hypergraphs are all linear in the number of rows, columns and nonzero entries of weight matrices in each layer. Hence, the complexity of generating hypergraph $H(\phi^k)$ for layer k can be given as $\theta(N + nnz(\mathbf{W}^k))$, where N and $\text{nnz}(\cdot)$ respectively denote the number of neurons per layer (i.e., the number of rows and columns of $\mathbf{W}^k$) and number of nonzero entries (i.e., connections) in a matrix.

5.1 Discussion

One challenge inherent in the parallel SGD algorithm is that processors perform communication between each consecutive layer, introducing a synchronization barrier. To alleviate synchronization overheads and improve the parallelization efficiency, input vectors can be processed in batches at each iteration (i.e., minibatch SGD can be performed instead of SGD). By simply modifying SpFF, batch processing can be enabled in such a way that instead of forwarding a single vector $\mathbf{x}^k$ between each consecutive layer, multiple vectors can be simultaneously processed in batches. That is, sparse matrix-matrix multiplications (SpMM) of the form $\mathbf{W}^k\mathbf{X}^{k-1}$ can be performed in each layer where $\mathbf{X}^{k-1}$ is formed by placing multiple $\mathbf{x}^{k-1}$ vectors as columns in $\mathbf{X}^{k-1}$. Hence, the main iteration of the inference step becomes $\mathbf{X}^k = f(\mathbf{W}^k\mathbf{X}^{k-1})$. The gradient vector $\boldsymbol{\delta}^L$ in the final layer is computed as the averages of gradients obtained over the vectors in the current batch. The SpBP algorithm is executed in the same way, since a single gradient vector is backpropagated to update weight parameters. Additionally, the proposed hypergraph partitioning is still applicable without any modifications, since the proposed model depends only on the DNN network structure.

The proposed hypergraph partitioning model can also be utilized for hybrid systems that provide both shared- and distributed-memory parallelism such as GPU or multiprocessor clusters. Implementations that utilize "MPI+CUDA" or "MPI+Openmp" can benefit from the proposed hypergraph partitioning approach to reduce communication costs between compute nodes that are connected by slower network connections. In this respect, our local SpMV computations can be replaced by more efficient libraries that utilize thread-level parallelism in multiprocessor and GPU architectures [4, 15]. Additionally, the proposed hypergraph models can also be utilized for heterogeneous computation systems by enforcing different target part weights to distribute different sized computational loads to processors.

The proposed hypergraph partitioning model and the SpMV-based SGD can also be utilized for convolution/pooling layers, which are widely utilized in popular convolutional neural network (CNN) architectures. These layers can be implemented as matrix-vector multiplications through constructing Toeplitz matrices [22], that capture convolution operation, and converting input data to vectors. Application of sparsification/pruning to CNNs induces sparsification on the corresponding Toeplitz matrices, making the proposed hypergraph model applicable to such cases.

6 EXPERIMENTS

6.1 Experimental Setup

We evaluate the performance of the proposed parallel SGD algorithm and hypergraph partitioning model on a benchmark provided by Sparse Deep Neural Network Graph Challenge[1] [36]. The benchmark uses synthetically generated sparse DNN models and MNIST database of handwritten digits [41]. These sparse networks are shown to be effective in terms of their training performance [37, 52]. We refer to the parallel training algorithm as H-SGD if the proposed hypergraph partitioning model is used to partition the neural networks. Otherwise, we refer to the algorithm as SGD to denote that random partitioning is utilized where neurons are assigned to processors uniformly at random in each layer. Random partitioning evenly splits weight matrices by assigning rows to processors uniformly at random and provides competitive computation/communication balance.

Sparse DNNs are generated by RadiX-Net synthetic sparse DNN generator [37] which takes two parameters: the number of layers and the number of neurons per layer. We used four different sized sparse DNNs consisting of 120 layers where numbers of neurons per

[1] https://graphchallenge.mit.edu/data-sets

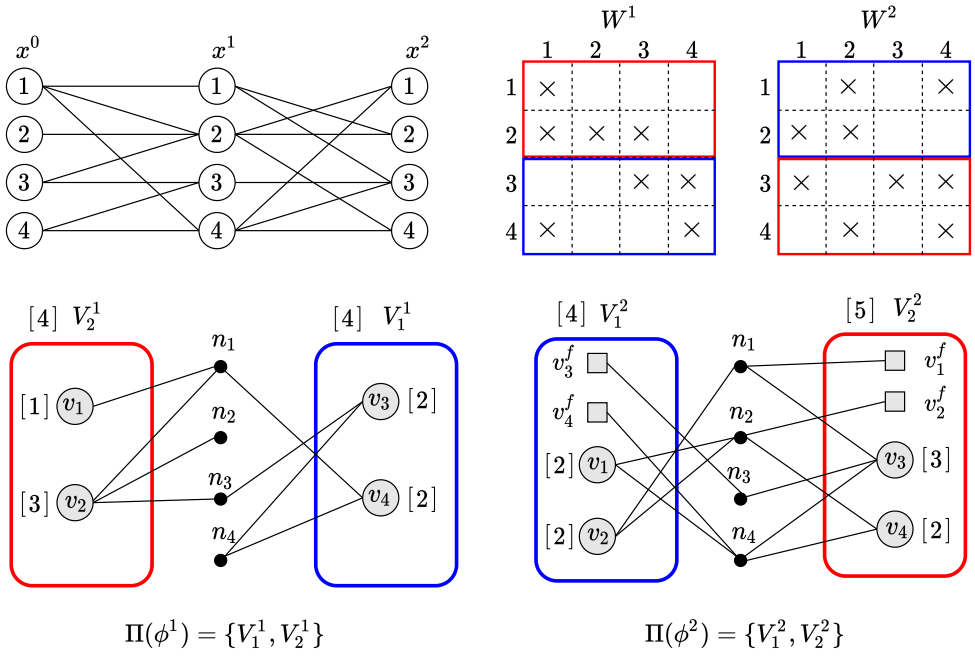

Figure 3: Top: A 3-layer sparse DNN and the corresponding weight matrices W^1 and W^2. Bottom: Hypergraphs $H(\phi^1)$ and $H(\phi^2)$ built for weight matrices. Partitions $\Pi(\phi^1)$ and $\Pi(\phi^2)$ induce a 2-way partitioning on weight matrices. $[\cdot]$ next to a vertex or a part shows the weight associated with that vertex and part, respectively.

layer are selected as $N = 1024, 4096, 16384$, and 65536, respectively. The MNIST database consists of 60,000 images of size 28×28 pixels and these images are scaled to 32×32, 64×64, 128×128 and 256×256. The scaled images are thresholded and flattened into 0-1 column vectors to be conformable with the input layers of sparse DNNs.

Running time experiments are performed on a high-performance computing system in which compute nodes are Lenovo NeXtScale nx360 M5 servers with 2×Intel Xeon E5-2630 v3 2.4 GHz (Haswell) 8 core processors (16 cores per node, 203 nodes, 3488 cores, 64GB DDR4 memory per node/4GB per core). The system provides at most 32 compute nodes (512 cores) to run our parallel codes. Compute nodes are connected via QLogic TrueScale InfiniBand. To test the effectiveness of the proposed hypergraph partitioning model as well as the scalability of the parallel SpMV-based training algorithm, we performed strong scaling experiments for H-SGD and SGD on numbers of processors $P = 32, 64, 128, 256$ and 512. Our SGD algorithm currently supports single-thread execution where we assign a single core to each MPI process and run a single thread per MPI rank. In our HPC system, the total memory of a compute node is not sufficient to store the whole DNN model for $N = 65536$ (Data-parallel approaches fail due to memory constraints). Therefore, our strong scaling experiments start from 32 cores (i.e., 2 nodes).

We implemented the parallel sparse SGD algorithm in C++ and implemented the inter-process communication operations via Message Passing Interface (MPI). We used sigmoid function as linear activation function $f(\cdot)$ and mean squared error as loss function $\mathbf{J}$. Initial connection weights of sparse DNNs are chosen uniformly at random from the interval $[-1, 1]$ and the learning rate is set to $\eta = 0.01$. The proposed hypergraph model is partitioned by using Patoh [7] where the maximum allowed imbalance ratio is set to $\epsilon = 0.01$ in each layer.

Algorithms that only perform inference computations on sparse DNNs [4, 15, 27, 49, 51] are not applicable in our general experimental setting. The best performing sparse DNN inference algorithms are generally designed for GPU-based systems and adopt data-parallelism. In these solutions, the backpropagation phase and weight update operations are not implemented. Data-parallel SGD solutions independently process input vectors in parallel and can not parallelize the computations associated with a single input vector, which limits the scalability by the batch size in training. Our solution achieves model-wise parallelism and can process a single input vector in parallel. Due to this fundamental difference of objectives and functionalities, we omit comparison against data-parallel solutions. To the best of our knowledge, our SGD solution is the first parallel SpMV-based training algorithm that achieves model-wise parallelism to train sparse DNNs on high-performance computing systems.

6.2 Performance Results

Table 1 compares the performance of SGD and H-SGD in terms of the communication volume and message counts metrics which relate to bandwidth and latency overheads of parallelization. The table displays both the average and maximum volume/number of messages sent by a processor for comparison of the average and maximum values. For each P, the first row displays the ratios of the respective values attained by H-SGD to those by SGD, whereas the

Table 1: Performance comparison of SGD and H-SGD

		1024					4096				
		Volume		Messages			Volume		Messages		
P		Avg	Max	Avg	Max	imb	Avg	Max	Avg	Max	imb
32	H	0.34	0.34	0.96	0.96		0.22	0.22	0.94	0.94	
		50	52	7	7	1.01	130	134	7	7	1.01
	R	149	154	7	7	1.05	594	603	7	7	1.04
64	H	0.31	0.32	0.82	0.83		0.23	0.23	0.93	0.93	
		29	31	12	12	1.01	84	87	14	14	1.01
	R	94	98	15	15	1.05	375	388	15	15	1.05
128	H	0.29	0.28	0.54	0.55		0.23	0.23	0.75	0.76	
		15	16	13	14	1.01	48	50	23	23	1.01
	R	53	57	24	25	1.08	212	222	30	30	1.08
256	H	0.39	0.36	0.49	0.48		0.20	0.20	0.42	0.43	
		9	10	11	12	1.03	23	24	21	22	1.03
	R	23	26	22	24	1.17	113	119	51	52	1.17
512	H	0.62	0.53	0.64	0.58		0.25	0.25	0.33	0.34	
		6	6	9	9	1.05	12	13	15	17	1.05
	R	10	12	14	16	1.24	47	52	46	49	1.24
		16384					65536				
32	H	0.17	0.17	0.92	0.93		0.15	0.15	0.91	0.91	
		407	412	7	7	1.01	1,439	1,454	7	7	1.01
	R	2,365	2,377	7	7	1.05	9,419	9,454	7	7	1.05
64	H	0.16	0.16	0.92	0.93		0.13	0.13	0.91	0.91	
		240	245	14	14	1.01	786	796	14	14	1.01
	R	1,491	1,512	15	15	1.05	5,938	5,973	15	15	1.05
128	H	0.15	0.16	0.90	0.90		0.12	0.13	0.91	0.91	
		130	134	27	27	1.01	417	424	27	28	1.02
	R	842	859	30	30	1.08	3,355	3,386	30	30	1.08
256	H	0.15	0.15	0.64	0.65		0.12	0.12	0.87	0.88	
		69	71	39	40	1.03	216	221	53	53	1.03
	R	448	462	61	61	1.17	1,786	1,820	61	61	1.17
512	H	0.14	0.14	0.32	0.33		0.12	0.12	0.57	0.58	
		32	34	33	34	1.05	109	112	69	70	1.05
	R	231	243	103	105	1.24	922	944	122	122	1.24

second and third rows display actual values. In the table, the last column shows the computational imbalance where the computational load is computed as the number of floating-point operations.

As seen in Table 1, on all processor counts, H-SGD incurs 38–71%, 75–80%, 83–86% and 85–88% less average/total communication volume for sparse DNNs with $N = 1024, 4096, 16384$ and 65536, respectively. Similarly, H-SGD incurs 47–72%, 75–80%, 83–86% and 85–88% less maximum send volume. The decrease in the bandwidth-related costs increases as the size of DNNs increases. The average and maximum communication volumes of processors are close to each other which denotes that communication balance is also achieved via the hypergraph partitioning.

In terms of the message count metrics, H-SGD achieves 4–51%, 6–67%, 8–68% and 9–43% smaller average message counts and 4–52%, 6–66%, 7–67% and 9–42% smaller maximum message counts. As the number of processors increases, the performance gap between the message count metrics of H-SGD and SGD increases

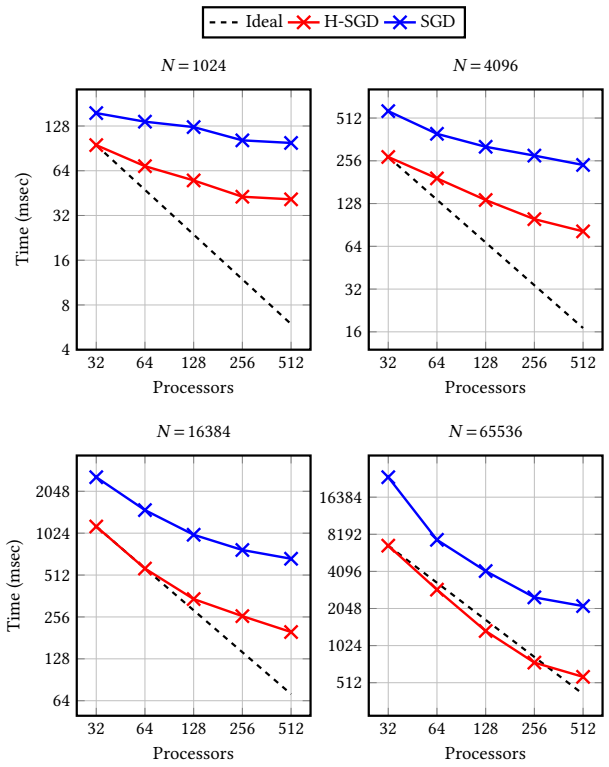

Figure 4: Strong scaling of SGD and H-SGD on different sized sparse DNNs consisting of 120 layers, and the number of neuron per layer $N = 1024, 4096, 16384$ and 65536, respectively.

in favor of H-SGD. In terms of computational load balance, H-SGD provides consistently better performance than SGD. These results demonstrate the effectiveness of the proposed hypergraph partitioning model since both the bandwidth- and latency-related costs are considerably minimized. Moreover, as the number of layers in sparse DNNs increases, performance improvement of the hypergraph partitioning model is expected to be higher due to optimizations achieved in each layer.

Figure 4 shows strong scaling of SGD and H-SGD. On each processor count, running times are measured as the average time required to process an input vector by H-SGD and SGD, where the averages are taken over 10^3 randomly selected input vectors. For all processor counts, H-SGD considerably improves the parallelization efficiency and runs 2.01–2.37x, 1.97–2.96x, 2.10–3.39x and 2.88–3.37x faster than SGD on sparse DNNs with $N = 1024, 4096, 16384$ and 65536, respectively. The best speedup is achieved on $N = 65536$ and $P = 512$ where H-SGD runs 3.39x faster than SGD. H-SpBP achieves the ideal speedup up to 512 processors on DNN with $N = 65536$.

The synchronization barrier due to the communication operations between successive layers constitutes the main source of latency overheads of the parallel SGD algorithm. As seen in Figure 4, the efficiency of parallel SGD algorithm considerably improves with the increasing number of neurons per layer, since latency overheads are considerably amortized on larger networks. Additionally, the

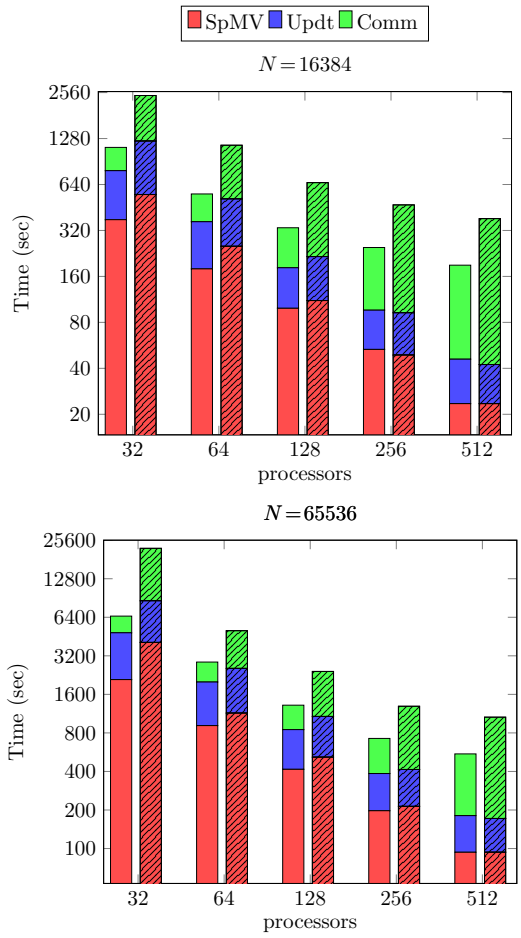

Figure 5: Breakdown of running time of H-SGD (solid) and SGD (tiled). "SpMV" corresponds to time spent on local sparse matrix-vector multiplications."Updt" corresponds to the time spent on gradient update operations."Comm" corresponds time spend for communication operations.

performance improvement achieved on running time by hypergraph partitioning increases with the increasing sizes of DNNs as well as the increasing number of processors.

In Figure 5, to better analyze the effects of the hypergraph partitioning on the performance of SGD, we break down the total time spent on communication and computation. As seen in the figure, the proportion of communication time to the overall running time increases with the increasing number of processors, whereas the proportion of time spent on local SpMV and gradient update computations decrease together with the total running time. For example, when $N = 65536$, the proportion of communication time respectively increases from 26% to 67% and 40% to 80% for H-SGD and SGD as the number of processors increases from $p = 32$ to $p = 512$. Hence, the improvements of hypergraph partitioning on the communication costs become more significant on the overall running time on larger processor counts. As the number of processors increases, the ratio of improvement in communication time to the improvement in the overall execution time gradually increases from 48% to 82% and 50% to 80% for $N = 16384$ and $N = 65536$, respectively. This can be attributed to the fact that on larger processor counts, communication costs become more dominant on the overall parallelization overheads and optimizations achieved by hypergraph partitioning on communication volume and message count metrics considerably improves.

We also observe that the hypergraph partitioning improves the performance of local SpMV and gradient update computations. Specifically, H-SGD reduces the running time of local computations by 1.9–2.7x on all processor counts as compared to SGD. The performance improvement on the local computations arises because hypergraph partitioning consistently achieves better computational balance and temporal cache-locality than random partitioning. The hypergraph partitioning assigns weight matrix rows, that are accessing similar input vector entries, to the same processor, which provides temporal cache locality in accessing input vector entries during local SpMV and gradient update computations. We refer the reader to [2] for a detailed explanation of how temporal cache-locality is achieved.

Table 2: Throughput results

		H-SpFF	GB	
Neurons	Layers	Throughput	Throughput	Speedup
1024	120	4.90E+10	7.11E+10	0.69
	480	5.41E+10	8.55E+10	0.63
	1920	5.57E+10	8.89E+10	0.63
4096	120	3.87E+10	7.38E+10	0.52
	480	3.71E+10	8.58E+10	0.43
	1920	3.63E+10	8.70E+10	0.42
16384	120	8.20E+10	5.13E+10	1.60
	480	7.91E+10	5.60E+10	1.41
	1920	7.81E+10	5.61E+10	1.39
65536	120	9.01E+10	2.80E+10	3.21
	480	8.57E+10	2.85E+10	3.01
	1920	8.55E+10	2.85E+10	3.00

6.3 Inference-only Computations

For inference-only computations, we enhanced SpFF by implementing local sparse matrix operations via SuiteSparse:GraphBLAS library [14]. The enhanced SpFF implementation supports batch processing and multi-thread execution. We also use the proposed hypergraph partitioning model and hence, we refer to SpFF as H-SpFF here. We compare H-SpFF against a data-parallel solution (GB) [15], that became one of the Graphchallange 2019 champions. GB utilizes SuiteSparse:GraphBLAS library to achieve shared-memory parallelism and is able to run on a single compute node. Similar to GB, H-SpFF processes all input vectors in a single batch.

Table 2 compares throughput values achieved by H-SpFF and GB for all sparse DNN configurations. Throughput corresponds to the ratio of the number of input vectors times the number of connections in a DNN divided by the execution time (i.e., number of

Table 3: Partitioning times (secs)

P	1024	4096	16384	65536
32	2.48	10.93	52.61	344.79
64	3.41	12.57	63.09	355.03
128	3.89	13.46	67.46	387.56
256	4.97	16.77	71.59	408.48
512	5.63	20.85	77.91	423.17

edges processed per second). The best throughput values of H-SpFF are measured on 512 cores with 128 MPI processes where we assign 4 cores for each MPI process and run 4 threads per MPI rank. We run GB on a single node in our local HPC system where the last two columns in the table display throughput and the relative speedup values measured on our local system. Standard nodes' memories were not enough for GB; hence we used fat nodes, which are in less number, that contain the same CPU configuration with higher memory.

As seen in Table 2, H-SpFF performs slightly worse than GB for small networks, whereas its performance considerably improves for larger networks, providing higher speedup values. For network configurations with $N = 16384, 65536$ and $L = 120$, H-SpFF achieves 1.6x and 3.2x speedups over GB, respectively. This can be attributed to the fact that the latency overheads introduced by the synchronization barrier between successive layers reduce the parallelization efficiency. The latency overheads are considerably amortized as the number of neurons per layer increases and the number of layers decreases. Therefore, H-SpFF is expected to perform better for network configurations with higher number of neurons and lower number of layers.

6.4 Partitioning Times

The preprocessing overhead of the partitioning is easily amortized, since the partitioning overhead is independent of the number of input vectors (i.e., training data size) fed into sparse DNNs, whereas the communication costs and the performance improvement attained by the hypergraph partitioning model increases with the increasing number of input vectors. Partitioning is performed once for each layer. Sets Xsend and Xrecv are computed in partitioning time and not modified hence do not affect the runtime. Table 3 displays partitioning times for $L = 120$ layer sparse DNNs we used in our experiments. As seen in the table, as the number of parts and the number of neurons per layer increases, partitioning times increase. Partitioning times are measured on a server with *2×Intel Xeon W-2245* 3.90GHz 8 core processors and 500GB DDR4 main memory.

7 CONCLUSION

We first introduced a distributed-memory parallel sparse DNN inference/training algorithm for high-performance computing systems. The solution is based on efficient parallelization of consecutive SpMV operations and achieves model-wise parallelism which significantly eliminates memory and bandwidth bottlenecks inherent in data-parallel approaches. We then proposed a novel hypergraph partitioning-based solution to address the latency overheads due to the communication operations between consecutive layers. The hypergraph partitioning model considerably improves communication overheads by reducing the total communication volume and the number of messages between processors while satisfying computational balance. Extensive experiments suggest that the proposed model-wise parallel solution scales to large processor counts especially when the proposed hypergraph partitioning is utilized. With the increasing number of neurons per layer and decreasing number of layers, latency overheads between consecutive layers are considerably amortized. Therefore, in cases where the whole DNN model can not fit into main memory and the data-parallel approaches are not feasible, the model-wise parallel inference/training algorithm and hypergraph partitioning model offer a feasible alternative for distributed memory systems.

8 ACKNOWLEDGMENTS

Computing resources used were provided by The Scientific Computing Research Technology Platform[2] at University of Warwick.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).
[2] Kadir Akbudak, Enver Kayaaslan, and Cevdet Aykanat. 2013. Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication. *SIAM Journal on Scientific Computing* 35, 3 (2013), C237–C262.
[3] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 193–205.
[4] Mauro Bisson and Massimiliano Fatica. 2019. A GPU Implementation of the Sparse Deep Neural Network Graph Challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.
[5] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[6] Adrián Castelló, Manuel F Dolz, Enrique S Quintana-Ortí, and José Duato. 2019. Analysis of model parallelism for distributed neural networks. In *Proceedings of the 26th European MPI Users' Group Meeting*. 1–10.
[7] Umit V Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on parallel and distributed systems* 10, 7 (1999), 673–693.
[8] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. 2020. Tifl: A tier-based federated learning system. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 125–136.
[9] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 571–582.
[10] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K Panda. 2020. Nv-group: link-efficient reduction for distributed deep learning on modern dense gpu systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–12.
[11] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *International conference on machine learning*. 1337–1345.
[12] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of machine learning research* 12, Aug (2011), 2493–2537.
[13] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).
[14] Timothy A Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.

[2] https://warwick.ac.uk/research/rtp/sc/

[15] Timothy A Davis, Mohsen Aznaveh, and Scott Kolodziej. 2019. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via SuiteSparse: GraphBLAS. In *2019 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[18] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574* (2019).

[19] Tong Geng, Tianqi Wang, Chunshu Wu, Chen Yang, Wei Wu, Ang Li, and Martin C Herbordt. 2019. O3BNN: An out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning. In *Proceedings of the ACM International Conference on Supercomputing*. 461–472.

[20] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[21] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks* 18, 5-6 (2005), 602–610.

[22] Robert M Gray. 2006. *Toeplitz and circulant matrices: A review.* now publishers inc.

[23] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse DNN models without hardware-support via tile-wise sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[24] Babak Hassibi and David G Stork. 1993. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*. 164–171.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[26] B Hendrickson and TG Kolda. [n.d.]. Partitioning Rectangular and Structurally Nonsymmetric Sparse Matrices for Parallel Processing, submitted to SIAM Journal of Scientific Computing.

[27] Mert Hidayetoğlu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-mei Hwu. 2020. At-Scale Sparse Deep Neural Network Inference With Efficient GPU Implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[28] Sara Hooker, Aaron Courville, Yann Dauphin, and Andrea Frome. 2019. Selective Brain Damage: Measuring the Disparate Impact of Model Pruning. *arXiv preprint arXiv:1911.05248* (2019).

[29] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

[30] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924* (2018).

[31] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *SysML 2019* (2019).

[32] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. 2016. How to scale distributed deep learning? *arXiv preprint arXiv:1611.04581* (2016).

[33] George Karypis. 1998. hMETIS 1.5: A hypergraph partitioning package. *http://www. cs. umn. edu/˜ metis* (1998).

[34] Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[35] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Albert Reuther, Ryan Robinett, and Sid Samsi. 2020. GraphChallenge. org Sparse Deep Neural Network Performance. *arXiv preprint arXiv:2004.01181* (2020).

[36] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. 2019. Sparse deep neural network graph challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[37] Jeremy Kepner and Ryan Robinett. 2019. RadiX-Net: Structured Sparse Matrices for Deep Neural Networks. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 268–274.

[38] Tamara G Kolda. 1998. Partitioning sparse rectangular matrices for parallel processing. In *International Symposium on Solving Irregularly Structured Problems in Parallel*. Springer, 68–79.

[39] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[41] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[42] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

[43] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[44] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. [n.d.]. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment* 13, 12 ([n. d.]).

[45] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[46] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 806–814.

[47] Christos Louizos, Max Welling, and Diederik P Kingma. 2017. Learning Sparse Neural Networks through L_0 Regularization. *arXiv preprint arXiv:1712.01312* (2017).

[48] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2019. Multithreaded Layer-wise Training of Sparse Deep Neural Networks using Compressed Sparse Column. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[49] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2020. Studying the effects of hashing of sparse deep neural networks on data and model parallelisms. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[50] Lin Ning and Xipeng Shen. 2019. Deep reuse: streamline CNN inference on the fly via coarse-grained computation reuse. In *Proceedings of the ACM International Conference on Supercomputing*. 438–448.

[51] Filip Pawłowski, Rob H Bisseling, Bora Uçar, and AN Yzelman. 2020. Combinatorial Tiling for Sparse Neural Networks. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[52] Ameya Prabhu, Girish Varma, and Anoop Namboodiri. 2018. Deep expander networks: Efficient deep networks from graph theory. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 20–35.

[53] Gerald Schubert, Georg Hager, Holger Fehske, and Gerhard Wellein. 2011. Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+ OpenMP programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 1751–1758.

[54] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[55] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[56] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1701–1708.

[57] Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. 2020. FFT-based Gradient Sparsification for the Distributed Training of Deep Neural Networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 113–124.

[58] Xiaoyun Wang, Zhongyi Lin, Carl Yang, and John D Owens. 2019. Accelerating DNN Inference with GraphBLAS and the GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[59] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*. 1299–1309.

[60] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. 2011. Fast sparse matrix-vector multiplication on GPUs: implications for graph mining. *arXiv preprint arXiv:1103.2405* (2011).

[61] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* 6 (2017).

[62] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2019. Fast deep neural network training on distributed systems and cloud TPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 11 (2019), 2449–2462.

[63] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. In *Advances in neural information processing systems*. 685–693.

[64] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).